Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

Implementation of the Ficus Replicated File System*

Richard G. Guy, John S. Heidemann, Wai Mak,
Thomas W. Page Jr., Gerald J. Popek[†] and Dieter Rothmeier
{guy,johnh,waimak,page,popek,dieter}@cs.ucla.edu

Department of Computer Science
University of California Los Angeles

Abstract

As we approach nation-wide integration of computer systems, it is clear that file replication will play a key role, both to improve data availability in the face of failures, and to improve performance by locating data near where it will be used. We expect that future file systems will have an extensible, modular structure in which features such as replication can be "slipped in" as a transparent layer in a stackable layered architecture. We introduce the Ficus replicated file system for NFS and show how it is layered on top of existing file systems.

The Ficus file system differs from previous file replication services in that it permits update during network partition if any copy of a file is accessible. File and directory updates are automatically propagated to accessible replicas. Conflicting updates to directories are detected and automatically repaired; conflicting updates to ordinary files are detected and reported to the owner. The frequency of communications outages rendering inaccessible some replicas in a large scale network and the relative rarity of conflicting updates make this optimistic scheme attractive.

Stackable layers facilitate the addition of new features to an existing file system without reimplementing existing functions. This is done in a manner analogous to object-oriented programming with inheritance. By structuring the file system as a stack of modules, each with the same interface, modules which augment existing services can be added transparently. This paper describes the implementation of the Ficus file system using the layered architecture.

1 Introduction

The Ficus project at UCLA is investigating very large scale distributed file systems. We envision a transparent, reliable, distributed file system encompassing a million hosts geographically dispersed across the continent, perhaps around the globe. Any host should be able to access any file in the distributed system with the ease that local files are accessed.

A large scale distributed system displays several critical characteristics: it is subject to continual partial operation, global state information is difficult to maintain, and heterogeneity exists at several levels. A successful large scale distributed file system must minimize the difficulties that these characteristics imply.

The scale of such a distributed system implies that the system will never be fully operational at any given time. For a variety of technical, economic, and administrative reasons various system components such as hosts, network links, and gateways will at times be unusable. Partial operation is the normal, not exceptional, status of this environment; new approaches are needed to provide highly available services to such a system's clients.

Large scale also prevents most nodes from

*This work was sponsored by DARPA under contract number F29601-87-C-0072.

[†]This author is also associated with Locus Computing Corporation.

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

attempting to maintain information about the global state of the system. (Imagine a filesystem table with millions of entries.) Even hosts with sufficient storage resources can not effectively track all the changes that occur across the distributed system, either because the changes are too rapid or communication is unreliable between the source of the change and the monitor.

Very large scale also implies that a high degree of hardware, software, and administrative heterogeneity exists. Software that can provide the desired availability must be easily utilized by a wide variety of existing host environments; it must also be tunable to meet both technical and administrative concerns. New tools must be sufficiently modular to allow easy attachment to existing services, and yet still provide acceptable performance.

These issues led us to explore the application and integration of several concepts to large scale file systems: stackable layers, file usage locality, data replication, non-serializable consistency, and dynamic volume locating and grafting.

Stackable layers: The stackable layers paradigm is used by Ritchie [16] as a model for implementing the stream I/O service in System V UNIX.[1] A stackable layer is a module with symmetric interfaces: the syntactic interface used to export services provided by a particular module is the same interface used by that module to access services provided by other modules in the stack. A stack of modules with the same interface can be constructed dynamically according to the particular set of services desired for a specific calling sequence.

We have found this model to be useful in designing and constructing file systems, as it allows easy insertion of additional layers providing new services. We have used it to provide file distribution and replication; we expect to use it for performance monitoring, user authentication and encryption.

File usage locality: The importance of modularity and portability implied that our replication service build on top of the existing UNIX file system interface. At least one previous attempt to adopt this philosophy abandoned it in the face of poor performance [19]. More recent studies of general purpose (university) UNIX file usage [6, 5] indicate a strong degree of file reference locality, and that appropriate caching methodologies can exploit this behavior to reduce file access overhead. The Ficus file system design takes advantage of these locality observations to avoid much of the overhead previously encountered in building on top of an existing UNIX file system implementation.

Replication: Data replication is used to combat the partial operation behavior that tends to degrade availability in large scale file systems. Each host may store one or more physical replicas of a logical file; clients are generally unaware which replica services a file request. The replication techniques used in Ficus are intellectual descendants of those used in the Locus [15] distributed operating system.

Non-serializable consistency: Most data replication management policies proposed in the literature adopt some form of serializability as the definition of correctness. The requisite mutual exclusion techniques to enforce serializability typically display an inverse relationship between update availability and read availability: ensuring high read availability forces a low update availability.

Ficus incorporates a novel, non-serializable correctness policy, ***one-copy availability***, which allows update of any copy of the data, without requiring a particular copy or a minimum number of copies to be accessible. One-copy availability is used in conjunction with automatic update propagation and directory reconciliation mechanisms.

One-copy availability provides strictly greater availability than primary copy [2], voting [21], weighted voting [7], and quorum consensus [10]. Our directory reconciliation mechanism tolerates a larger class of concurrent non-serializable updates than the replicated "dictionaries" of [4, 1, 22]. The replicated directory techniques in [3, 18] are based on quorum consensus, and thus also have lower availability. The Deceit file system [20] allows partitioned update without a quorum, but has no mechanism for reconciling concurrent updates to replicas of a single directory.

Volume locating and grafting: Locating a particular file in a very large scale distributed system requires a robust, distributed mechanism. Dynamic movement of files must be supported

[1] UNIX is a trademark of AT&T.

Appeared in the Proceedings of the Summer USENIX Conference, Anaheim, CA, June 1990, pages 63-71

without requiring any sort of advance global agreement. Ficus incorporates a volume autograft mechanism along with a segmented, distributed, replicated graft table.

The remainder of this paper describes key architectural details of the Ficus file system as of April, 1990. Further discussion of the ideas touched on above can be found in [13, 9, 8].

2 Ficus layered design

The Ficus layered file system model comprises two separate layers constructed using the vnode interface. NFS is employed as a transport mechanism between remotely located Ficus layers, and can also be used as a means for non-Ficus hosts to access Ficus file systems. Figure 1 shows the general organization of Ficus layers; the NFS layer is omitted when both layers are co-resident.

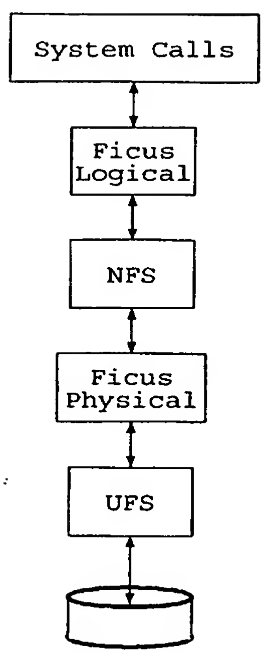

Figure 1: Ficus Stack of Layers

2.1 Vnode interface

The single most important design decision to be made when using the stackable layers paradigm is the definition of the interface between layers. Ideally, the interface will be general enough to allow for later extensibility in unplanned directions. The streams interface [16], for example, is remarkably simple and general: messages may be placed on an input queue for processing by the layer. Each layer dequeues and processes messages of types it recognizes; unrecognized message types are passed on to the next layer in the sequence.

Interface definitions can also be more closely tailored to the particular application area, as is the case with the vnode [12] interface used in SunOS for file system management. The vnode interface is defined by a set of about two dozen services, together with their calling syntax and parameters. In SunOS, the vnode interface is used to hide details of particular file system implementations, including the location (local or remote) of the actual file storage.

We adopted the vnode interface for stackable layers in Ficus, with some misgivings. Leveraging an existing interface for file system modules is clearly beneficial when getting started. The vnode interface is also in widespread use, so persuading others to add Ficus modules to existing implementations is much easier than introducing an entirely new interface. On the other hand, the vnode interface is quite rigid: adding services desired by new layers encountered a variety of difficulties, of which several are mentioned below.

Using the vnode interface also allows Ficus to utilize existing UFS (UNIX File System) and NFS (Network File System) [17] services in SunOS in critical ways. For example, Ficus can use the UFS as its underlying nonvolatile storage service, which means Ficus is not burdened with the details of how best to physically organize disk storage. Ficus is also able to use NFS as its remote access and transport mechanism, again relieving Ficus of substantial work.

While the Ficus layers are conceptually organized as in Figure 1, each is implemented as a new virtual file system type, as indicated in Figure 2.

Appeared in the Proceedings of the Summer USENIX Conference, Anaheim, CA, June 1990, pages 63-71

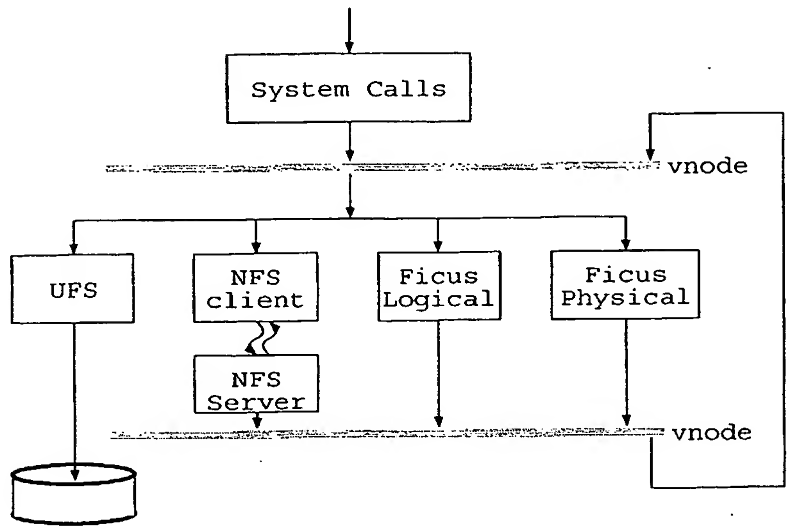

Figure 2: Layered Architecture Using Vnodes

2.2 NFS as a transport layer

NFS is essentially a host-to-host transport service with a vnode interface. Generally speaking, then, any layer that uses a vnode interface can be unaware whether the immediately adjacent functional layers are local, or perhaps remote and accessed via an intervening NFS layer. The Ficus replication service layers are able to use NFS for transparent access to remote layers, without having to build a transport service.

Unfortunately, the NFS implementation in SunOS does not fully preserve vnode semantics. The stateless philosophy of NFS clashes occasionally with vnode semantics, and the resulting NFS implementation is not simply a "host-to-host transport service with a vnode interface". For example, the vnode services OPEN and CLOSE are not supported by the NFS definition, and so are ignored: a layer intending to receive an OPEN will never get it if NFS is in between.

NFS also incorporates optimizations intended to reduce communications and improve performance. The file block caching and directory name lookup caching are not fully controllable (e.g., there is no user-level way to disable all caching), which results in unexpected behavior for layers which are not able to adopt the assumptions inherent in the NFS cache management policies.

2.3 Adding new vnode services

The Ficus replication service employs functionality not anticipated (understandably) by the vnode interface design. Rather than add several new services outside the vnode framework (as in Deceit [20]) we chose to overload existing vnode services. This maximizes portability, at a slight expense of interpreting an overloaded service and perhaps limiting its use in some way.

For example, Ficus is able to use effectively the open/close information that NFS intercepts and ignores, so a new service is required. We overloaded the LOOKUP service by encoding an open/close request as a null-terminated ASCII string of sufficient length to be passed on by NFS without interpretation or interference.[2]

2.4 Cooperating layers

Layers can be added to a stackable design singly or in groups. Layers inserted as a group may be stacked together or separated by other, existing

[2] The reduction in the maximum length of a file name component from 255 to about 200 does not seem to be a significant loss: we've never seen a component of even length 40.

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

layers. For example, the Ficus replication service is composed of two layers, a *logical file layer* and a *physical replica layer*. These layers are separated by an NFS layer when the logical and physical layers are on different hosts.

2.5 Ficus Logical layer

The Ficus logical layer presents its clients (normally the UNIX system call family) with the abstraction that each file has only a single copy, although it may actually have many physical replicas. The logical layer performs concurrency control on logical files, and implements a replica selection algorithm in accordance with the consistency policy in effect. The default policy of one-copy availability is to select the most recent copy available.

The logical layer also oversees update propagation notification and automatic reconciliation of directory replicas. When a logical layer requests a physical layer to update a file or directory, an asynchronous multicast datagram is sent to all available replicas informing them that a new version of a file may be obtained from the replica receiving the update. Each physical layer reacts to the update notification as it sees fit: it may propagate the new version immediately, or wait for some later, more convenient time.

Periodically, a logical layer invokes a file and directory reconciliation mechanism to compare file replica subtrees. The details of the reconciliation algorithms are beyond the scope of this paper; see [9, 8] for further information.

Ficus files are organized in a general DAG of directories; unlike UNIX, Ficus directories may have more than one name.[3] The logical layer maps a client-supplied name into a Ficus file handle, which contains a set of fields that uniquely identify the file across all Ficus systems. The Ficus file handle is used to communicate file identity between the logical and physical layers.

[3] This characteristic is a consequence of the ability to change the name of a directory while some copies are unavailable. When non-communicating directory replicas are concurrently given new names, it is often later necessary to retain multiple names.

2.6 Ficus Physical layer

The Ficus physical layer implements the concept of a file replica. Each Ficus file replica is stored as a UFS file, with additional replication-related attributes stored in an auxiliary file. (These attributes would be placed in the inode if we were to modify the UFS.) Ficus uses the version vector technique of [14] to detect concurrent unsynchronized updates to files.

Ficus directories are stored as UFS files, not UFS directories. A Ficus directory entry maps a client-specified name into a Ficus file handle, which then must be mapped into an inode by the UFS. This second mapping is implemented by encoding the Ficus file handle into a hexadecimal string used by the UFS as a pathname.

The dual-mapping nature of the current Ficus implementation is difficult to implement efficiently [19], but is not inherently expensive. The on-disk file organization closely parallels the logical Ficus name space topology, which allows the existing UFS caching mechanisms to continue to exploit the strong directory and file reference locality observed in [6, 5]. We believe the unacceptable performance observed by [19] in a similar dual-mapping scheme used in a prototype of the Andrew File System occurred because the lower level name mapping was incompatible with the locality displayed at higher levels.

3 Replication

Ficus incorporates data replication as a primary technique for achieving a high degree of availability in an environment characterized by communications interruptions. Each file and directory in a Ficus file system may be replicated, with the replicas placed at any set of Ficus hosts.

3.1 Basics

A logical file is represented by a set of physical replicas. Each replica bears a *file identifier* that globally uniquely identifies the logical file, and a *replica identifier* that uniquely identifies that particular replica. The logical layer uses a file handle composed (in part) of file identifier and replica identifier to communicate with physical layers about a file.

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

The number and placement of file replicas is effectively unbounded.[4] A client may change the location and quantity of file replicas whenever a file replica is available.

Associated with each file replica is a *version vector*[14] which encodes the update history of the replica. Version vectors are used to support concurrent, unsynchronized updates to file replicas managed by noncommunicating physical layers.

3.2 Update notification/propagation

Updates are initially applied to a single physical replica. The invoking logical layer notifies other physical layers managing replicas of the updated file that a newer version exists in the updated replica. A physical layer that receives an update notification makes an entry for the file in a *new version cache*. An update propagation daemon consults this cache to see what new replica versions should be propagated in, and performs the propagation when it deems it appropriate to expend the effort. Rapid propagation enhances the availability of the new version of the file; delayed propagation may reduce the overall propagation cost when updates are bursty.

For regular files, update propagation is simply a matter of atomically replacing the contents of the local replica with those of a newer version remote replica. Ficus contains a single-file atomic commit service to support file update propagation. A shadow file replica is used to hold the new version until it is completely propagated, and then the shadow atomically replaces the original by changing a low-level directory reference. If a crash occurs before the shadow substitution, the original replica is retained during recovery and the shadow discarded.[5]

Update propagation for directories is more difficult because of the side effects of directory update: files may be allocated, reference counts adjusted, and so on. Simply copying directory contents is incorrect; in a sense, a directory operation needs to be "replayed" at each replica. In Ficus, a *directory reconciliation* algorithm is used for this purpose.

3.3 Reconciliation

A reconciliation algorithm examines the state of two replicas, determines which operations have been performed on each, selects a set of operations to perform on the local replica which reflect previously unseen activity at the remote replica, and then applies those operations to the local replica.

The Ficus directory reconciliation algorithm [9] determines which entries have been added to or deleted from the remote replica, and applies appropriate entry insertion or deletion operations to the local replica. The standard set of UNIX directory operations is supported.

The directory reconciliation algorithm used for update propagation and the basic file update propagation service are both incorporated into the general Ficus file system reconciliation protocol. This protocol is executed periodically to traverse an entire subgraph (not just a single node), and reconcile the local replica against a remote replica. The execution proceeds concurrently with respect to normal file activity, so that client service is not blocked or impeded.

4 Volumes

Ficus uses volumes[6] as a basic structuring tool for managing disjoint portions of the file system. Ficus volume replicas are dynamically located and grafted (mounted) as needed, without global searching or broadcasting. The tables used for locating volume replicas are replicated objects similar to directories, and are managed by the same reconciliation algorithms used for directory replicas.

[4] There is a current limit of 2^{32} replicas of a given file, and 2^{32} logical layers.

[5] Note that this commit service is not necessary for the correct operation of the general Ficus functionality. While its performance impact is usually small, it can have a significant effect if the client is updating a few points in a large file. To avoid alteration of the UFS, rewriting the entire file is necessary. That cost could, of course, be avoided by putting a commit function into the storage layer.

[6] Ficus volumes are similar to Andrew [11] volumes; both decouple the logical concept of subtree from the physical storage details in order to support flexible volume "replica" placement. Ficus does not require a replicated volume location database.

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

4.1 Basics

The Ficus file system[7] is organized as a directed acyclic graph of *volumes*. A volume is a logical collection of files that are managed collectively. Files within a volume typically share replication characteristics such as replica location and the number of replicas.

A volume is represented by a set of *volume replicas* which function as "containers" in which file replicas may be placed. The set of volume replicas forms a maximal, but extensible, collection of containers for file replicas. A volume replica may contain at most one replica of a file, but need not store a replica of any particular file.

A volume replica is stored entirely within a UNIX disk partition. The mapping between volume replicas and disk partitions is determined by the host providing the storage. Many volume replicas may be stored in a single partition; no relationship between volume replicas is implied by placement in disk partitions.

A volume is a self-contained[8] rooted directed acyclic graph of files and directories. A volume's boundaries are the root node at the top, and volume *graft points* at the bottom. The volume root is normally a directory; a graft point is a special kind of directory, as explained below. Each volume replica must store a replica of the root node; storage of all other file and directory replicas is optional.

4.2 Identifiers

A volume is uniquely named internally by a pair of identifiers: an *allocator-id*, and a *volume-id* issued by the allocator. Prior to system installation, each Ficus host is issued a unique value as its allocator-id; for example, an Internet host address would suffice. Individual volume replicas are further identified by their *replica-id*, so a volume replica is globally uniquely identified by the triple ⟨allocator-id, volume-id, replica-id⟩.

Within the context of a particular volume, a logical file is uniquely identified by a *file-id*. A particular file replica is then identified by appending the replica-id of the containing volume replica to the file-id, as in ⟨file-id, replica-id⟩. A fully specified identifier for a file replica is ⟨allocator-id, volume-id, file-id, replica-id⟩; this identifier is unique across *all* Ficus hosts in existence.

Each volume replica assigns file identifiers to new files independently. To ensure that file-ids are uniquely issued, a file-id is prefixed with the issuing volume replica's replica-id. A file-id is actually, therefore, a tuple ⟨replica-id, unique-id⟩.

[7] We use *file system* (two words) to refer to a particular type of file service, e.g., UNIX file system or VMS file system. A *filesystem* (one word) is a self-contained portion of a UNIX file system normally one-to-one mapped into a single disk partition.

[8] Directory references do not cross volume boundaries.

4.3 Graft points

A graft point is a special file type used to indicate that a (specific) volume is to be transparently grafted at this point in the name space. Grafting is similar to UNIX filesystem mounting, but with a number of important differences. The particular volume to be grafted onto a graft point is fixed when the graft point is created, although the number and placement of volume replicas may be dynamically changed.

A graft point is very similar to a regular directory. It can be renamed or given multiple names. A graft point is itself replicated; a graft point replica is contained in a particular volume replica.

Many graft points for a particular volume may exist, even within a single volume. The resulting organization of volumes would then be a directed acyclic graph and not simply a tree.

A graft point contains a unique volume identifier and a list of volume replica and storage site address pairs. Therefore, a one-to-many mapping exists between a graft point replica and the volume replicas which can be grafted on it. Each graft point replica may have many volume replicas grafted at a time.

The list of volume replicas and the (Internet) addresses of the managing Ficus physical layers are conveniently maintained as directory entries. Overloading the directory concept in this way allows implicit use of the Ficus directory reconciliation mechanism to manage a replicated object (a graft point) with similar semantics and syntactic details.

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

4.4 Autografting

When the Ficus logical layer encounters a graft point while translating a pathname, a check is made to see if an appropriate volume replica is already grafted. If not, the information in the graft point is used to locate and graft the volume replica of interest.

A Ficus graft is very dynamic: a graft is implicitly maintained as long as a file within the grafted volume replica is being used. A graft that is no longer needed is quietly pruned at a later time.

5 Development methodology

The stackable layers paradigm extends to our development methodology. The vnode interface normally accessible only inside the kernel has been "exposed" to the application level through a set of *vnode system calls*, so that a functional layer can execute at the application level. The standard NFS server already provides a channel for a kernel layer to utilize a vnode layer in another address space; we customized a copy of the NFS server daemon code to run outside of the kernel as the interface to the Ficus layers.

This approach allows us to use application level software engineering tools to develop and test outside of the kernel what will ultimately be kernel level service layers. The performance penalty for crossing address space boundaries complicates performance measurements and analysis, but otherwise the methodology has proven sound.

The goal has been to provide a programming environment at the application level that is the same as a kernel-based module would experience. Today, Ficus layers may be compiled for application level or kernel resident execution merely by setting a switch.

Our hope had been that once application level debugging was complete, correct kernel-based execution would be automatic. That has not been achieved, in part because of the single threaded application environment we set up, and because of other minor differences. Nevertheless, the ability to operate outside the kernel that was made so easy by the stackable architecture and exposure of vnode services, markedly shortened development and testing time.

6 Performance notes

Ficus is in use at UCLA for normal operation. Its perceived performance is good, but an extensive evaluation is still under way. The major potential performance costs that are observed result from two considerations: execution overhead from crossing multiple formal layer boundaries that might not be present in a more monolithic structure, and additional I/Os from maintenance of needed attribute information. The actual cost of crossing a layer boundary is low – one additional procedure call, one pointer indirection, and storage for another vnode block. In the current implementation, the increased I/O cost can be noticeable, however.

The Ficus physical layer design and implementation accrues additional I/O overhead when opening a file in a non-recently accessed directory. Four I/Os beyond the normal UNIX overhead occur: an inode and data page for the underlying UNIX directory and an auxiliary replication data file must be loaded from disk, as well as the Ficus directory inode and data page. (The last two correspond to normal UNIX overhead.) Opening a recently accessed file or directory involves no overhead not already incurred by the normal UNIX file system.

7 Conclusions

Our experience with the approach described in this paper has been quite positive. The modularity provided by stackable layers, as well as the simplicity in design and implementation afforded by the optimistic reconciliation approach has been especially significant.

The stackable architecture appears to work quite well: layers can indeed be transparently inserted between other layers, and even surround other layers. A replication service can be added to a stack of "vnode" layers without modifying existing layers, and yet perform well.

The vnode interface is not ideal; a more extensible interface is desired. An inode level interface to files and extensible directory entries would allow us to avoid implementing Ficus directories on top of the UNIX directory service; extensible inodes would allow us to dispense with auxiliary files to store replication data. With these changes

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

virtually all additional I/O overhead over standard UNIX and NFS would be eliminated.

The availability of a general reconciliation service was also very useful. Usually, one must deal with the many boundary and error conditions that occur in a distributed program with a considerable variety of cleanup and management code throughout the system software. Instead, in Ficus failures may occur more freely without as much special handling to ensure the integrity and consistency of the data structures environment. Reconciliation service cleans up later. For example, volume grafting was made considerably easier by the (easy) transformation of its necessarily replicated data structures into Ficus directory entries. No special code was needed to maintain their consistency.

In sum, we are optimistic that services such as those provided by Ficus will be of substantial utility generally, and easy to include as a third-party contribution to a user's system.

References

[1] James E. Allchin. A suite of robust algorithms for maintaining replicated data using weak consistency conditions. In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 562–570, October 1976.

[3] Joshua J. Bloch, Dean Daniels, and Alfred Z. Spector. Weighted voting for directories: A comprehensive study. Technical Report CMU-CS-84-114, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[4] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[5] Rick Floyd. Directory reference patterns in a UNIX environment. Technical Report TR-179, University of Rochester, August 1986.

[6] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report TR-177, University of Rochester, March 1986.

[7] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*. ACM, December 1979.

[8] Richard G. Guy. *Ficus: A Very Large Scale Reliable Distributed File System*. Ph.D. dissertation, University of California, Los Angeles, 1990. In preparation.

[9] Richard G. Guy and Gerald J. Popek. Reconciling partially replicated name spaces. Technical Report CSD-900010, University of California, Los Angeles, April 1990. Submitted concurrently for publication.

[10] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, February 1986.

[11] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[12] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Conference Proceedings*, pages 238–247, Atlanta, GA, Summer 1986. USENIX.

[13] Thomas W. Page, Jr., Gerald J. Popek, Richard G. Guy, and John S. Heidemann. The Ficus distributed file system: Replication via stackable layers. Technical Report CSD-900009, University of California, Los Angeles, April 1990. Submitted concurrently for publication.

[14] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on*

Appeared in the Proceedings of the Summer USENIX Conference,
Anaheim, CA, June 1990, pages 63-71

Software Engineering, 9(3):240–247, May 1983.

[15] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture*. Computer Science Series, The MIT Press, 1985.

[16] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[17] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119–130. USENIX, June 1985.

[18] Sunil Sarin, Richard Floyd, and Nilkanth Phadnis. A flexible algorithm for replicated directory management. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 456–464. IEEE, June 1989.

[19] M. Satyanarayanan et al. The ITC distributed file system: Principles and design. *Operating System Review*, 19(5):35–50, 1 December 1985.

[20] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Cornell University, November 1989.

[21] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the 16th IEEE Computer Society International Conference*. IEEE, Spring 1978.

[22] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *ACM Symposium on Principles of Distributed Computing*, August 1984.